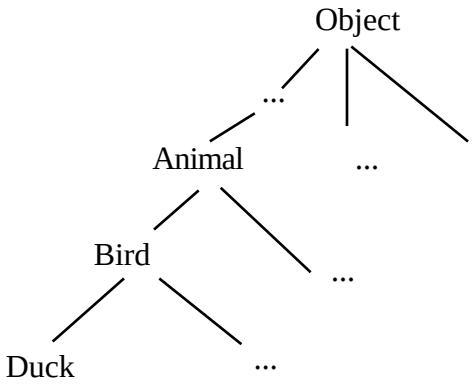
OOP2 -- Inheritance

Modularity and encapsulation are the most important parts of OOP. OOP also includes a significant inheritance component. But first, a word of warning. Inheritance is a neat and intellectually appealing bit of technology. However, true uses of inheritance are somewhat rare. Encapsulation remains the dominant theme in OOP, while inheritance comes in now and then for certain problems.

Hierarchy

Classes in OOP are arranged in a tree-like hierarchy. A class' "superclass" is the class above it in the tree. The classes below a class are its "subclasses." The semantics of the hierarchy are that classes have all the properties of their superclasses. In this way the hierarchy is general up towards the root and specific down towards its leaves. The hierarchy helps add logic to a collection of classes. It also enables similar classes to share properties through "inheritance" below. A hierarchy is useful if there are several classes which are fundamentally similar to each other. In C++, a "base class" is a synonym for superclass and "derived class" is a synonym for subclass.



Inheritance

"Inheritance" is the process by which a class inherits the properties of its superclasses. Methods and instance variables are inherited. When an object receives a message, it checks for a corresponding method in its class. If one is found, it is executed. Otherwise the search for a matching method travels up the tree to the superclass of the object's class. This means that a class automatically responds to all the messages and has all the storage of its superclasses.

Overriding

When an object receives a message, it checks its own methods first before consulting its superclass. This means that if the object's class and its superclass both contain a method for a message, the object's method takes precedence. In other words, the first method found in the hierarchy takes precedence. This is known as "overriding," because it gives a class an easy way to intercept messages

before they get to its superclass. Most OOP languages implement overriding based on the run-time class of objects. In C++, run-time overriding is an option invoked with the "virtual" keyword.

Polymorphism

A big word for a simple concept. Often, many classes in a program will respond to some common message. In a graphics program, many of the classes are likely to implement the method "drawSelf()." In the program, such an object can safely be sent the drawSelf() message without knowing its exact class since all the classes implement or inherit drawSelf(). In other words, you can send the object a message without worrying about its exact class and be confident that it will just do the right thing depending on its class.

Inheritance Vocabulary

OOP Hierarchy
Superclass / Subclass
Inheritance
Overriding
ISA -- the subclass ISA instance of the superclass -- it has **all** the properties that instances of the superclass are supposed to have (and it has some additional properties as well)

Inheritance Warning

Inheritance is a clever and appealing technology.
However, it is only usable in somewhat rare circumstances -- where you have several similar classes.
It is a common error for beginning OOP programmers to try to use inheritance for everything.
Modularity may be less flashy, but it is incredibly common. Inheritance is rare, but for certain problems, it is a great solution.

Horse / Zebra Example

With inheritance, we define classes in terms of other classes. This can be a great shortcut if the classes are similar.
Suppose you have a hierarchy of all the animals, except the zebra was omitted and you have been asked to add it in.
No: define the zebra from scratch
Yes: locate the Horse class. Introduce Zebra as a subclass of Horse
    Zebra inherits 90% of its behavior (no coding required)
    In the Zebra class, define the few things that are features of Zebras but not Horses

Grad Variation

Suppose we want to add a Grad class based on the Student class -- Grad students are like students, but with two differences...
    * Years on thesis -- a grad has a count of the number of years worked on thesis
    - getStress() is different -- Grads are more stressed. Their stress is (2* the Student stress) + yearsOnThesis

Student Inheritance

Student defined by int units
Grad is everything that a Student is + the idea of yearsOnThesis (yot)
"isa" relationship with its superclass -- Grad isa Student
Subclass has **all** the properties of its superclass + a few
Grad overrides getStress() with a specialized version

General vs. Specific

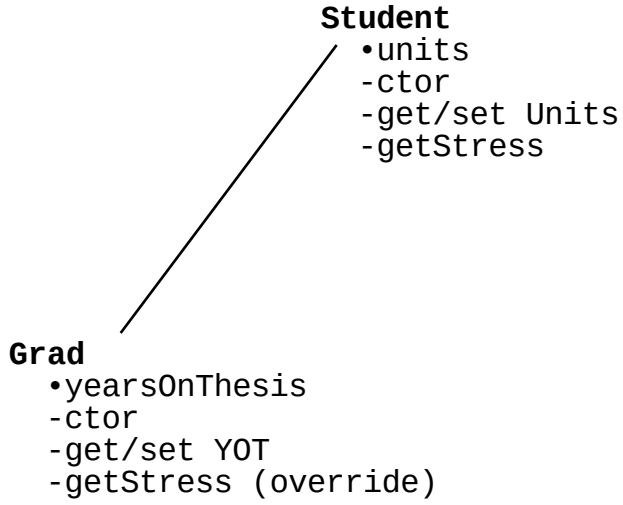
Grad (subclass) has more properties, is more constrained, is more specific compared to Student

Student (superclass) has fewer properties, is less constrained, is more general compared to Grad

Student/Grad Design Diagram

The following is a good sort of diagram to make when thinking about an OOP inheritance design. Plan the division of responsibility between a superclass and subclass.

('•' = instance variable, '-' = method)



Simple Inheritance Client Code

```
Student s = new Student(10);
Grad g = new Grad(10, 2);  // ctor takes units and yot
s.getStress();          // (100) goes to Student.getStress()
g.getUnits();           // (10) goes to Student.getUnits() -- INHERITANCE
g.getStress();          // (202) goes to Grad.getStress() --OVERRIDING
```

Never Forget Class

In Java, no matter what code is being executed, the receiver never forgets its class.

EG in the above g.getUnits() example, the code executing against the receiver in the Student class, but the receiver knows that it is a Grad

Semantics of "Student s;"

What does a declaration like "Student s;" mean in the face of inheritance?

NO: "s points to a Student object"

YES: "s points to an object that responds to all the messages that Students respond to"

YES: "s points to a Student, or a subclass of Student"

OOP Pointer Substitution

A subclass object can be used in a context which calls for a superclass object
This works because of the ISA property -- Grad ISA Student
Therefore, a pointer to a Grad object be assigned to Student pointer.

```
Student s = new Student(10);
Grad g = new Grad(10);
s = g;        / / ok -- subclass may be used in place of superclass
/ / what operations are allowed on s?
```

The reverse is not allowed however

```
Student s = new Student(10);
Grad g = new Grad(10);
g = s;        / / NO, does not compile
```

Compile Time -- Error Check

The compiler will only allow code where it is100% clear that the receiver responds to the given message.
This is all based on the compile time type system -- the declared types of ivars and other variables in the source code.
Because of the substitution rule, the compile time and run time type systems diverge somewhat.
The compile time type system is more loose -- not knowing the exact class of the receiver.
e.g., with the following method, the compiler, only knows that "s" points to either a Student object or a Grad object

```
void foo(Student s) {
  / / s points to Student or Grad -- don't know for sure
```

Run Time

In Java, the run time type system is exact -- the receiver knows exactly what class it is.
The run time type system is used to resolve message sends (i.e. "message / method resolution").

Substitution Code

```
Student s = new Student(10);
Grad g = new Grad(10, 2);
s = g;   / / ok
s.getStress();  / / (202) ok -- goes to Grad.getStress() (overriding)
s.getUnits();   / / (10) ok -- goes to Student.getUnits (inheritance)
s.getYearsOnThesis();        / / NO -- does not compile (s is compile time type Student)
```

Downcast

The programmer m,ay place a cast in the code to give the compiler more specific type information

In the above example, the compiler knows that the variable s is at least Student, but does not know the stronger (more specific) claim of type Grad.

We could put a (Grad) cast around s at the end of the above example like this...

```
((Grad)s).getYearsOnThesis();
```

This does not permanently change the s variable, the cast is just a part of that expression

This is known as a "downcast", since it makes the more specific, stronger claim, which is in the down direction in the typical super / sub class diagram

In Java, all casts are checked at run time, and if they are not correct, they throw a ClassCastException.

In C++, if a cast turns out at run time to be wrong, horrible random crashing tends to result (The dynamic_cast operator attempts to work around this problem)

Student / Grad Memory Layout

Implementation detail: in memory, the ivars of the subclass are layered on top of the ivars of the superclass

Result:: if you have a pointer to the base of an instance of the superclass (Grad), you can treat it as if it were a superclass object (Student) and it just works since the objects look the same from the bottom up. Again: you can treat an instance of the superclass as if it were the subclass, and it works ok. A Grad object looks like a Student object.

Inheritance Client Code

The compiler works with the compile-time type system, and only allows message sends that are guaranteed to work at runtime. The programmer can put in casts to edit the compile-time types of expressions.

At run-time, the message-method resolution usees the run-time type of the receiver, not the compile-time type -- this is a feature. Some languages use compile-time types for message resolution, but that coding style is very unintuitive..

```
Student s = new Student(10);
Grad g = new Grad(15, 2);
Student x = null;

System.out.println("s " + s.getStress());
System.out.println("g " + g.getStress());

// Note how g responds to everything s responds to
// with a combination of inheritance and overriding...
g.dropClass(3);
System.out.println("g " + g.getStress());
```

```
/*
 OUTPUT...
   s 100
   g 302
   g 242
*/

// s.getYearsOnThesis();   // NO does not compile
g.getYearsOnThesis();   // ok

// Substitution rule -- subclass may play the role of superclass
x = g;   // ok

// At runtime, this goes to Grad.getStress()
// Point: message/method resolution uses the RT class of the receiver,
// not the CT class in the source code.
// This is essentially the objects-know-their-class rule at work.
x.getStress();

// g = x;   // NO -- does not compile,
// substitution does not work that direction

// x.getYearsOnThesis();   // NO, does not compile

((Grad)x).getYearsOnThesis(); // insert downcast
// Ok, so long as x really does point to a Grad at runtime
```

isIrate() Example

Suppose we have an isIrate() method in the Student class that returns true if the receiver has a stress over 100.

(In Java, messages that do a boolean test on the receiver tend to start with the word "is".)

Question: how does this work if we send the isIrate() message to a Grad object?

```
public boolean isIrate() {
 return(getStress() > 100);
// POPS DOWN to Grad.getStress()
// if the receiver is a Grad
}
```

Client code...

```
Student s = new Student(...);
Grad g = new Grad(....);
s.isIrate();          // does the right thing
g.isIrate();          // does the right thing
```

g.isIrate() Series

Where does the code flow go when sending isIrate() to a Grad object?

1. Student.isIrate()
2. Grad.getStress() / / pop-down
3. Student.getStress() / / the super.getStress() call in Grad.getStress

"Pop-Down" Rule

The receiver knows its class
The flow of control jumps around different classes
No matter where the code is executing, the receiver knows its class and does message->method mapping correctly for each message send.
e.g. Receiver is the subclass (Grad), executing a method up in the superclass (Student), a message send that Grad overrides will "pop-down" to the Grad definition (getStress())

super.getStress()

The "super" keyword is used in methods and ctors to refer to code up in the superclass.
In the Grad code, the message send "super.getStress();} means...
- Send the getStress() message
- In the message / method resolution, do not use the getStress() method in the Grad class.
- Instead, search for a matching method beginning with the superclass, Student.

This syntax is necessary so that an override method, such as getStress(), can still refer to the original version up in the superclass.
Often, an override method is not written from scratch. Intead, it is built on the superclass version.
In C++, a method can be named at compile time by its class, e.g. Student.getStress(), but there is no equivalent in Java. "Super" does the regular runtime message / method search, but begins the search one class higher.

Subclass Ctor

The subclass needs a constructor.
The ctor should take as arguments the data needed for itself, and also any arguments needed by the superclass ctor.
On its first line, the ctor should call "super(...);" -- this special syntax calls the superclass ctor to initialize that part of the object.
If no super ctor is specified, the default ctor will be called.
Every class needs its own ctor complete with all its arguments spelled out. The ctor must be present, even if it just one line that calls the superclass ctor. In this sense, ctors are not inherited -- the subclass has to define its own.

this Ctor

One ctor in a class can call another ctor in that class using "this" on the first line.
In the following code, the default ctor calls the 2-argument ctor

```
public Grad() {
   this(10, 0);
}

public Grad(int units, int yot) {
   ...
}
```

instanceof

Java includes an instanceof operator that may be used to check the run time type of a pointer...

```
if (x instanceof Grad) { ...
```

instanceof with a **null** pointer always returns false

Using instanceof is regarded as a possible sign of poor OOP design. Ideally, the message / method resolution selects the right piece of code depending on the class of the receiver, so further if / switch logic with instanceof should not be necessary.

Grad.java

```
// Grad.java

/*
 Grad is a subclass of Student -- a simple example of subclassing.
 -adds the state of yearsOnThesis.
 -overrides getStress() to provide a Grad specific version.
*/
public class Grad extends Student {

   private int yearsOnThesis;


   /*
    Ctor takes an initial units and initial years on thesis.
   */
   public Grad(int units, int yearsOnThesis) {
      // NOTE "super" must be first if used --
      // chains up to the superclass constructor
      super(units);

      // NOTE use "this" since ivar and param have same name --
      // there is not wide agreement if that's good style or not.
      this.yearsOnThesis = yearsOnThesis;
   }


   /*
    Default ctor builds a Grad with 10 units and 0 yot.
   */
   public Grad() {
       this(10, 0);    // "this" on first line calls
                       // a different ctor in the same class
   }


   /*
    Override
    Grad stress is 2 * Student stress + yearsOnThesis.

    NOTE: avoid code repetition between subclass/superclass
    at all costs -- that's why we use Student.getStress()
    for the core of our computation.
   */
```

```
public int getStress() {
   // NOTE "super" still invokes message/method resolution
   // but it starts the search one class higher up
   // (there is no super.super)
   int stress = super.getStress();

   return(stress*2 + yearsOnThesis);
}

// Standard accessors
public void setYearsOnThesis(int yearsOnThesis) {
   this.yearsOnThesis = yearsOnThesis;
}

public int getYearsOnThesis() {
   return(yearsOnThesis);
}


public static void main(String[] args) {
   Student s = new Student(10);
   Grad g = new Grad(15, 2);
   Student x = null;

   System.out.println("s " + s.getStress());
   System.out.println("g " + g.getStress());

   // Note how g responds to everything s responds to
   // with a combination of inheritance and overriding...
   g.dropClass(3);
   System.out.println("g " + g.getStress());

   /*
    OUTPUT...
      s 100
      g 302
      g 242
   */

   // s.getYearsOnThesis();   // NO does not compile
   g.getYearsOnThesis();   // ok

   // Substitution rule -- subclass may play the role of superclass
   x = g;   // ok

   // At runtime, this goes to Grad.getStress()
   // Point: message/method resolution uses the RT class of the receiver,
   // not the CT class in the source code.
   // This is essentially the objects-know-their-class rule at work.
   x.getStress();

   // g = x;   // NO -- does not compile,
   // substitution does not work that direction

   // x.getYearsOnThesis();   // NO, does not compile

   ((Grad)x).getYearsOnThesis(); // insert downcast
   // Ok, so long as x really does point to a Grad at runtime
```

```
   }


   /*
    Example .equals() method in the Grad class --
    true if two Grad objects have the same state.
   */
   public boolean equals(Object other) {
      // Common optimization for == case
      if (this == other) return(true);

      // Is the other object the right class?
      // (instanceof is false for null)
      if (!(other instanceof Grad)) return(false);

      // Look inside the other object
      // (example of sibling access)
      Grad grad = (Grad)other;
      return(grad.units==units && grad.yearsOnThesis==yearsOnThesis);
   }

}

/*
 Things to notice...

 -The ctor takes both Student and Grad state -- the Student state is passed up
 to the Student ctor by the first "super" line in the Grad ctor.

 -getStress() is a classic override. Note that it does not _repeat_ the code
  from Student.getStress(). It calls it using super, and fixes the result.
  The whole point of inheritance is to avoid code repetition.

 -Grad responds to every message that a Student responds to -- either
 a) inherited such as getUnits()
 b) overriden such as getStress()

 -Grad also responds to things that Students do not,
 such as getYearsOnThesis().
*/
```

Inheritance / Notification Style

Here is an illustration of how all this inheritance stuff is actually used...
Suppose there is a Car class with go(), stop(), and turn() methods
Suppose there is an existing system of code that sends "notifications" to the car over time: go(), stop(), go(), ...
You want to create your own car, but that turns differently, it beeps every time it turns, ...
  Subclass off Car
  Override the turn() method with your own definition that beeps, and then calls super.turn()
The existing system / car relationship with the methods go(), stop(), etc. continues to work

A turn() notification will pop down to use your turn(), and then pop back up and continue using the standard car code

Notification Style Results

Use this style as a way of integrating your code with library code -- subclass off a library class, 90% inherit the standard behavior, and 10% override a few key methods.

e.g. Collections -- subclass off AbstractList to inherit addAll(), toString(), .. and lots of other methods. It pops down to use your implementation of the core functions add(), iterator(), ...

e.g. Servlets -- inherit the standard HTTP Servlet behavior and define custom behavior in a few key method overrides.